1997

The Use of Prime Numbers as an Effective Method of Cryptology

Joshua Flynn
College of Saint Benedict/Saint John's University

The Use of Prime Numbers as an Effective Means of Cryptography

A THESIS

The Honors Program

College of St. Benedict/St. John's University

In Partial Fulfillment of the Degree of Bachelor of Arts
In the Department of Computer Science/Mathematics

by

Joshua J. Flynn

May, 1997

Title: The Use of Prime Numbers as an Effective Means of Cryptography

Approved by:

Project Advisor - Jennifer R. Galovich
Mathematics Dept: Assistant Professor

Department Reader - Dave Hartz
Mathematics Dept: Associate Professor

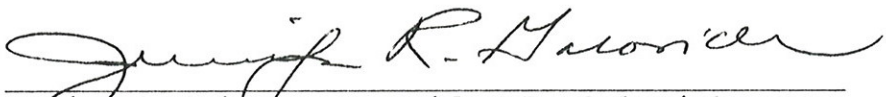
Department Reader - J. Andrew Holey
Computer Science Dept: Assistant Professor

Department Chair of Mathematics
Michael Gass

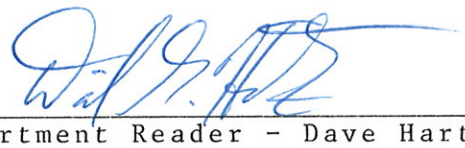
Department Chair of Computer Science
Lynn R. Ziegler

Director: Honors Thesis Program
Margaret Cook

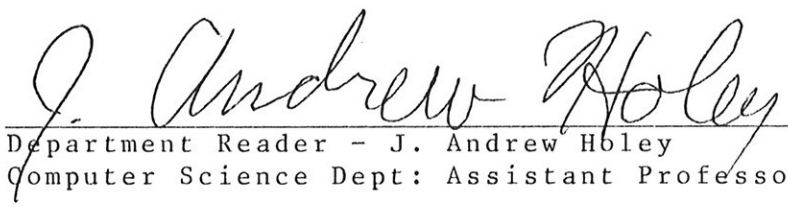
Director: Honors Program
Tony Cunningham



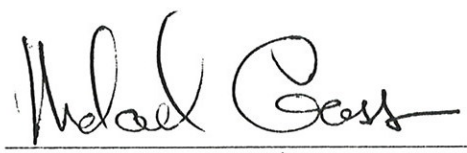
Project Advisor - Jennifer R. Galovich
Mathematics Dept: Assistant Professor



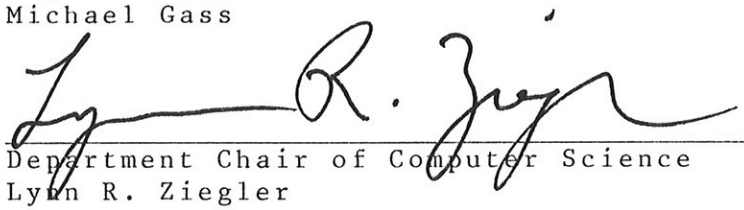
Department Reader - Dave Hartz
Mathematics Dept: Associate Professor



Department Reader - J. Andrew Holey
Computer Science Dept: Assistant Professor



Department Chair of Mathematics
Michael Gass



Department Chair of Computer Science
Lynn R. Ziegler

Table of Contents

The Use of Prime Numbers as an Effective Means of Cryptography

1.0 Introduction

In the age of technology, people have increased their use of computers and software packages such as Netscape to exchange information. On the Internet, it is now possible to trade securities (stocks and bonds), do personal banking, and purchase merchandise with credit cards. All of these actions require the exchange of very important personal information.

The use of information networks for business has expanded enormously in the past two decades. In a recent study, an average of $800 billion was transferred among partners in international currency markets every day, about $1 trillion transferred daily among U.S. banks, and another $2 trillion worth of securities traded daily in the New York markets (primarily the New York Stock Exchange) [U.S. Congress 1994b, 3].

Another place where important information is stored is in defense systems. Between April 1990 and May 1991, hackers penetrated computer systems of thirty-four Department of Defense sites by weaving their way through university, government and commercial systems on the Internet. The hackers exploited a security hole in the Trivial File Transfer Protocol, which allowed users on the Internet to access a file containing encrypted passwords without logging onto the system [U.S. Congress 1994b, 4]. With the added amount of personal information passing through these networks, it is important to make sure that this information is kept private from outsiders.

It is not hard to intercept messages that are traveling over a network. On any Ethernet or FDDI network, someone directly connected to the network can see the messages going over the line. Thus, it is possible for them to remove this information from the line to

read or change it. To get to its destination, a message may travel through many different networks, increasing the risk of an outsider stealing the information.

The main need for information security lies in the dangerous consequences that could result if information were to fall into the wrong hands. If a terrorist were able to get his hands on the sequence codes for a U.S. missile, a lot of destruction could happen. Also, people could literally be robbed without even knowing it if their credit card number were to be seen when purchasing merchandise over the Internet. Information must be protected so the preceding examples do not occur. We do this by making the information unreadable to outsiders, a process which is also known as cryptography.

But what makes cryptography secure? The amount of protection a cryptosystem (a particular protocol for cryptography) provides depends on how hard it is to break it, that is, to read the information. Once this is done, an outsider could manipulate the information to the owner's demise. However, with a secure cryptosystem, outsiders should not be able to break in easily without hours of relentless work, by which time the information could be obsolete.

There are many different types of cryptography, each with different levels of security. The question to ask is, what algorithm can be effective in maintaining a high level of protection from unauthorized access? This is answered by number theory, and more specifically, properties of prime numbers.

1.1 What is cryptography?

Cryptography, from the Greek *kryptos* meaning "hidden" and *graphein* meaning "to write", is the art and science of making communications unintelligible to all except the intended recipient(s) [Konheim 1981, 3].

In trying to understand what cryptography is, we must look at both sides of the picture. On one side there is Alice, the owner of the information, who does not want others to be able to access this information. On the other side is Bob, who wants to be able to access the private information. Cryptography is the means by which Alice makes her information private, keeping Bob from reading the information.

In this example, it may seem as if Bob is the bad guy, and Alice is the good guy. However, there are cases where the tables can be turned. Suppose Alice is a criminal or terrorist, and Bob is an FBI agent. Should Bob be able to access the information of Alice in order to protect the well being of society?

In cryptography, if we are able to obtain unauthorized access to information, then there is a need to strengthen the level of security used in the process of cryptography. This could include using a new cryptosystem, or upgrading the current one to add further protection. However, it is impractical that mathematical results of breaking into the systems and intercepting messages can go on forever. Thus, one might think that once all of the methods of defeating the security are found, and the security of the system is strengthened to protect against every method, then there will be no practical way to break into systems to access private information.

However, this cannot happen. In order to be able to decrypt a message, there must exist a way in which one can break the security. The most important strategy of an algorithm is to make the way out, or trapdoor, extremely difficult to find.

1.2 The history of cryptography

The origins of cryptography can be traced back nearly four millennia to the hieroglyphic writing system of the Egyptians [Konheim 1981, 3]. Ancient Hebrews

enciphered certain words in the scriptures. Also, one of the best known forms of cryptography was created nearly 2000 years ago when Julius Caesar used a simple substitution cipher, now known as the Caesar cipher (also known as the Caesar shift).

The Caesar shift algorithm is extremely simple, and involves a shift value *N*. In enciphering a message using this algorithm, we simply take a character and displace it by a value of *N*. For example, using a shift value of 5, encrypting the character "d" will yield "i" (e - f - g - h - i; a shift of five). Caesar's cipher may have been a good algorithm back then, but it is too insecure today.

In the time of Caesar cryptography was mainly used as a means of sending secret messages within the military. However, the importance of secrecy in those times was less important than it is today. Whereas secrecy in the time of Caesar was needed only in military combat, today it is important for many more types of information, ranging from the personal to the national level. Also, whereas in the time of Caesar, cryptography was mainly accomplished by hand calculations, today it is achieved by the use of computers, running information through programs in order to make it unreadable to outsiders.

Today an algorithm such as the Caesar cipher is called a classical cryptosystem because it involves either transposition or substitution of characters. Such cryptosystems can be either mono-alphabetic (one substitution/transposition used) or poly-alphabetic (several substitutions/transpositions used). These methods do not rely heavily on number theory such as modern cryptosystems.

2.0 Terms and definitions

Looking closer at the process of cryptography, we can see it as the transforming of a message P (called the *plaintext*) into another message C (called the *ciphertext*) by some function E such that $E(P) = C$ [Konheim 1981 ,1]. The function E is called the *enciphering* function, and the transforming of P by the function E is called the *enciphering* of message P.

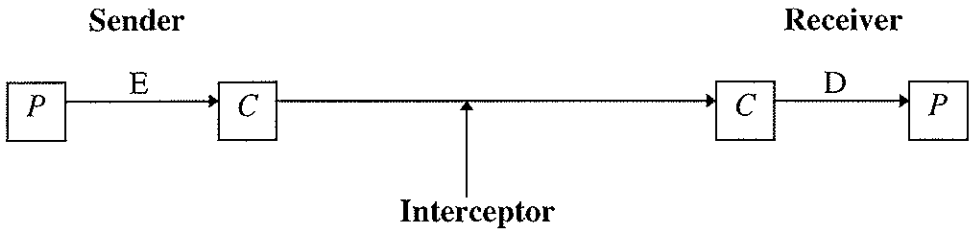
In addition to the enciphering function, cryptography also includes a method of recovering the message from the transformed message. The recovery of the original message P is done through another function D such that $D(C) = P$. The function D is called the *deciphering* function, and the transformation of C by the function D is called the *deciphering* of message P. Looking at this, we see that $D(E(P)) = P$, so the enciphering function E and the deciphering function D are inverse functions of one another.

Figure 1 shows the process of the message P being enciphered by the sender, sent to the receiver, and recovered through the deciphering process. This figure also shows where the outsider could intercept the message, provided that the message is being sent across an unsecured network.

Opposite of private key cryptography, where both the encryption and decryption keys are kept private, in public key cryptography the enciphering information is made private, while all other information is kept private. Public key methods allow for outside communication using the encryption algorithm. The deciphering information is kept private because if it is known by someone outside the group, a message can be deciphered by an unintended recipient who will be able to read and even change the information in

the message. In Figure 1, if the interceptor knew the deciphering key, and if he/she got hold of the message E(P), then he/she could easily compute D(E(P)) = P.

Figure 1: Layout of sending an encrypted message.



The functions E and D are usually mappings to and from finite groups such as $(\mathbf{Z}/n\mathbf{Z})^*$ or $\mathbf{F}_q^*$, which are groups whose operation is usually written multiplicatively.

In the remainder of this paper I discuss why there is a need for information security. Then, two different methods of cryptography which have been used to implement such security are explained, along with examples of how the methods work on sample data. Also, the advantages and disadvantages of each method are explained. Finally, I give an explanation of the programming requirements for the RSA algorithm, along with source code for a prototype of this algorithm.

3.0 The RSA model

One method of encryption which has received widespread attention is that of the RSA model. Named after its creators, R.L. Rivest, A. Shamir, and L. Adelman, the RSA algorithm uses the notion that it is very hard to find prime factors of large numbers, and provides a rather safe method of information security. I call it a "rather" safe method because the RSA method does have its problems, which I will describe later.

3.1 An explanation of RSA

In examples that are given throughout this paper, there are two users who are depicted as A (Alice) and B (Bob). These users are the sender and receiver of messages to and from one another using the algorithm that is being described. Values that are generated or computed by each specific user are labeled with their respective name (for example, n_A is a number that is generated or computed by A).

For the sake of simplicity, we will suppose there are only two users, A and B, and calculations will only be made for user A. First, A chooses two large prime numbers p_A and q_A (about 100 decimal digits each), and computes $n_A = p_A q_A$. The numbers p_A and q_A are kept secret by their owner A. Since A knows the factorization of her number n_A, she can easily compute $\varphi(n_A) = (p_A - 1)(q_A - 1) = n_A + 1 - p_A - q_A$, which is also kept secret to A[1]. A must now randomly choose an integer e_A between 1 and $\varphi(n_A)$ such that e_A is relatively prime to $\varphi(n_A)$ (in other words, that $\gcd(e_A, \varphi(n_A)) = 1$). The final number which A must compute is the multiplicative inverse of e_A modulo $\varphi(n_A)$: $d_A = e_A^{-1} \bmod \varphi(n_A)$. [Koblitz 1994, 92] We know that such a d_A exists because e_A is relatively prime to $\varphi(n_A)$.

Once all of these integers have been determined, Alice will have her encryption key (n_A, e_A) and decryption key (n_A, d_A). The encryption key is made public, while the decryption key is kept secret by its owner.

The enciphering of a message P is the mapping from $\mathbf{Z}/n_A\mathbf{Z}$ to itself defined by

---

[1] The symbol φ is the Greek letter Phi. $\varphi(n_A)$ represents the number of integers less than n_A and relatively prime to n_A.

$E(P) = P^{e_A} \bmod n_A = C$. An encrypted message can be deciphered by raising C to the d_Ath power modulo n_A. Since e_A and d_A are multiplicative inverses mod $\varphi(n_A)$, then $e_A d_A \equiv 1 \bmod \varphi(n_A)$. So, we have $\varphi(n_A) \mid e_A d_A - 1$, and thus $e_A d_A - 1 = k\,\varphi(n_A)$ for some integer k, so $e_A d_A = k\,\varphi(n_A) + 1$. Therefore, $P^{e_A d_A} = P^{k \cdot \varphi(n_A)} \cdot P^1 = [P^k]^{\varphi(n_A)} \cdot P^1 \equiv P \bmod n_A$[2]. Thus, if $\gcd(P, n) = 1$, then $C^{d_A} \bmod n_A = (P^{e_A})^{d_A} = P \pmod{n_A}$.

However, suppose that $\gcd(P, n) \neq 1$. Then let's use the assumption that if $p \mid n$ (p is prime) then $p - 1 \mid de - 1$. We can show that this holds in the case where $de \equiv 1 \bmod \varphi(n)$: Let's assume that $n = pq$ (so $p \mid n$) and $\varphi(n) = (p - 1)(q - 1)$. Since $de \equiv 1 \bmod \varphi(n)$, then $\varphi(n) = (p - 1)(q - 1) \mid (de - 1)$. By transitivity it follows that $(p - 1) \mid (de - 1)$.

So, we now need to show that $P^{de} = P \bmod n$. It suffices to show that $P^{de - 1} \equiv 1 \bmod n$. Again, we are assuming that $n = pq$ where p and q are prime, and that if $p \mid n$, then $p - 1 \mid de - 1$. We know by the second assumption that $de - 1 = i\,(p - 1)$ for some integer i, and therefore $P^{de - 1} = P^{i \cdot (p - 1)}$. By Fermat's Theorem, we know that $P^{i \cdot (p - 1)} \equiv 1 \bmod p$. So, $P^{de - 1} \equiv 1 \bmod p$, and likewise, $P^{de - 1} \equiv 1 \bmod q$. It follows that $p \mid P^{de - 1} - 1$ and $q \mid P^{de - 1} - 1$. Therefore, $pq \mid P^{de - 1} - 1$ since p and q are prime, and we can conclude that $P^{de - 1} \equiv 1 \bmod n$. Thus, for the multiplicative inverses e_A and d_A, we have $P^{e_A d_A \,(\bmod \varphi(n_A))} \bmod n_A = P^1 \bmod n_A = P \pmod{n_A}$.

However, before we encipher the message P, it must be converted to its numerical equivalent. In converting a message from alphanumeric to numeric form, we make calculations based on the size of the alphabet being used (N), in which each character is

[2] If $\gcd(P, n) = 1$ by Euler's generalization of Fermat's Theorem

given a numerical value. The entire message P is broken into blocks of the same size, say x; the last block of the message may contain fewer characters.

Next, each block in succession is changed to its numerical equivalent through an agreed upon formula. The values of the formula depend on the numeric values of each of the characters in the block. In our formula, the numerical equivalent of each character is multiplied by N^{x-k}, where $0 \leq x - k \leq x - 1$ and k is the location of the character in the block (e.g. the third character). For instance, using a 26 character alphabet, if a character has the numeric equivalent of 10 and is the third character in a five-character block, then the result would be $10 \cdot 26^{5-3}$ After calculating $M \cdot N^{x-k}$ for each character M in the block, all of the results are added together. The final result will be concatenated to the end of the result from the previous block behind padded zeroes to determine the start and end of blocks. When we have made these calculations for every block, we have the ciphertext. [Koblitz 1994, 93].

Here is an example of how the conversion works. Suppose you are using the standard 26 letter alphabet, with a = 0, b = 1, … , z = 25. Next, a block of letters, say "PRIME" (there is no case sensitivity), is converted by taking the numerical equivalent of each letter times N^w. The value of each character in the block is P = 15, R = 17, I = 8, M = 12, and E = 4. The numeric equivalent of this block of letters can be found by computing $15 \cdot 26^4 + 17 \cdot 26^3 + 8 \cdot 26^2 + 12 \cdot 26^1 + 4 \cdot 26^0 = 7159156$.

To show the entire process of the RSA algorithm, we will follow up with an example. Again, for the sake of simplicity we will use small integers in the example and will assume that the message is based only on the standard 26 letter alphabet.

Suppose Alice chooses her primes to be $p_A = 281$ and $q_A = 167$. Then, she computes $n_A = p_A q_A = 46927$ and $\varphi(n_A) = (p_A - 1)(q_A - 1) = 46480$. Then suppose that she chooses $e_A = 39423$. Finally, Alice computes $d_A = e_A^{-1} \bmod \varphi(n_A) = 26767$.

Since we are using the standard alphabet, then N = 26. In order for Bob to send the message "RSA" to Alice with the enciphering key $(n_A, e_A) = (46927, 39423)$, he must first convert "RSA" to its numerical equivalent, which is $17 \cdot 26^2 + 18 \cdot 26 + 0 = 11960$. This is the numerical value of P, and now must be enciphered by computing $E(P) = P^{e_A} \bmod n_A = 11960^{34923} \bmod 46927 = 45832$. The value calculated is the ciphertext, which Bob sends to Alice.

Since Alice knows the deciphering key $(n_A, d_A) = (46927, 26767)$, she is able to compute $D(C) = C^{d_A} \bmod n_A = 45832^{26767} \bmod 46927 = 11960$. After this value has been calculated, Alice then factors the result. She will first divide by 26^2, and then take the remainder and divide it by 26^1. The last remainder will be the final character. Therefore, Alice calculates $11960/26^2 = 17 = R$, $(11960 \bmod 26^2)/26 = 468/26 = 18 = S$, and $(468 \bmod 26) = 0 = A$. Putting the letters back together, we have the original plaintext message: "RSA".

3.2 Requirements and programming logic for implementing the RSA model

The first requirement is related to security issues. In generating the numbers e, p and q, which should be quite large, the user should use a random number generator. With 'random' numbers, it is most likely that the number will not have any meaning to the user (e.g. a birthdate, social security number, or vehicle identification numbers). The risk of an outsider finding the number is reduced because instead of finding easy information about the user which could relate to a value, they don't know where to start looking.

Another requirement that a random number generator helps to fulfill is that of efficiency. The random generation of values by a computer makes the computations much easier as well as more efficient in comparison to a user generating such large numbers by hand.

A program for generating large prime numbers p and q would entail the following process. First, a large odd number with a specific number of digits (specified by the user) is generated and tested to determine whether or not it is prime (or probably prime). If it is, then this is p. If not, increment the number by 2 and test its primality again. We repeat the process until a prime (or probably prime) number is found. The same is done for q, with the user specifying this integer to be a couple digits longer than p for security reasons. There are many different methods which can be used to determine whether or not a number is prime, some of which are described in section 5. When the prime numbers p and q have been generated, the user can then compute $n = pq$.

If a user were to write a program to perform such a function as generating e, he/she might use the following method. First of all, since $\varphi(n) = (p - 1)(q - 1)$, where p and q are both odd primes, then $\varphi(n)$ is always going to be even. Because of this, e must be odd; for if it were even, then $\gcd(e, \varphi(n)) \neq 1$. Therefore, in determining the value e, a large odd number would be generated that contains the number of digits that the user specifies. This number would then be checked to see if $\gcd(e, \varphi(n)) = 1$. If this is not the case, the number would be incremented by 2 and checked again until the condition is true. A number $e < \varphi(n)$ will always be returned, because $\gcd(\varphi(n) - 1, \varphi(n))$ will always equal 1.

3.3 Problems with the RSA method of cryptography

Even though RSA is thought to be a secure method of cryptography, it does have its flaws. First of all, in order to break this system, all we need to do is to factor n, finding p and q. Once these two numbers are known, $\varphi(n)$ can be computed, and using the publicly known enciphering key e, it is possible to compute the deciphering key d.

However, this is not a large problem, as it becomes very difficult to find prime factors of a number n when the number exceeds 130 digits (RSA was broken using a 130-digit n in April, 1996 using Pollard's number field sieve [Pomerance 1996, 1473]). Some methods that can be used to factor large numbers are described in section 6.

Another problem lies in the choices of the numbers p and q. When a user chooses these numbers, he/she should make sure that they are not too close together, perhaps one being a couple digits longer than the other. The reason for this is that if p and q are relatively close, one can easily factor n from the inside-out using Euler's method of factoring. These methods are described in detail in section 6.

Another problem that can occur with the RSA system as a result of an unlucky choice of p and q is that the RSA method can be broken without factoring n. It has been found that for certain keys, reciphering a message a small number of times will restore the original plaintext message. Thus, given a ciphertext $C_0 = P^e \pmod{n}$ and public key (e, n), a cryptanalyst may be able to determine P by computing $C_j = (C_{j-1})^e \pmod{n}$ for $j = 1, 2, \ldots$ until C_j is a repeat of the message. [Foss 1986, 15]

4.0 The Discrete Logarithm

It has been concluded that public key cryptosystems are relatively slow compared to classical cryptosystems, and thus it is often more realistic to use them in a limited role in conjunction with a classical cryptosystem in which the actual messages are transmitted. [Koblitz 1994, 98] For example, the discrete logarithm problem could be used as a means of key exchange for a classical cryptosystem. One such method of key exchange using the discrete log was developed by W. Diffie and M.E. Hellman.

Suppose we have a finite group $(\mathbf{Z}/n\mathbf{Z})^*$. For $b \in (\mathbf{Z}/n\mathbf{Z})^*$, one can compute b^x for large x very rapidly. If we are given an element $y \in (\mathbf{Z}/n\mathbf{Z})^*$, which we know is of the form b^x (assuming the base b is fixed), how can we find the power of b that gives y, i.e. how can we compute $x = \log_b y$? This is known as the discrete logarithm problem. [Koblitz 1994, 97]

4.1 The Diffie-Hellman key exchange system

Even though the discrete logarithm problem is not used very much in algorithms as a method of message transmission, it has been very useful as a method of exchanging keys (e.g. encryption and decryption keys). The basic outline of this algorithm is as follows:

1. Alice and Bob agree on two large integers, n and g, such that n is prime and $1 < g < n$. The values g and n do not have to be secret.
2. Alice chooses a random large integer x and computes $\mathbf{X} = g^x \bmod n$
3. Bob chooses a random large integer y and computes $\mathbf{Y} = g^y \bmod n$.
4. Alice sends $\mathbf{X}$ to Bob, and Bob sends $\mathbf{Y}$ to Alice. The values of x and y are kept secret by Alice and Bob, respectively.

5. Alice computes $k_A = Y^x \bmod n$.

6. Bob computes $k_B = X^y \bmod n$.

When the process is complete, the key $k_A = k_B = g^{xy} \bmod n$. This method of key exchange is secure because the only publicly known values are n, g, **X** and **Y**. Unless one can compute the discrete logarithm and recover x or y, this problem can't be solved. So, $k_A = k_B$ is the secret key that Alice and Bob computed independently [Schneier 1994, 29].

4.2 An example using the Diffie-Hellman key exchange system

This key exchange system can be used to exchange the key used in shift-key encryption. More specifically, the Diffie-Hellman key exchange system can be used by people who are implementing the shift-key encryption of single letter message units in the 26 letter alphabet N = P + B mod 26, where B is the message, P is the shift and N is the number of characters in the alphabet. The key exchange is used to determine how far to shift the letters in the message B. Small values for n, g, x, and y will be used to allow the reader to understand the calculations more clearly.

Suppose Alice and Bob want to send a message to one another using shift-key encryption. They must agree on a value (or key) to shift the letters in the message without letting others know what the shift-key is. However, before they can agree on a key, they must first agree on the two integers g and n.

Let's assume that Bob and Alice agree on the values $n = 67$ and $g = 2$. Thus the finite group which they will be working with is $\mathbf{F}_{67}^* = \{1, ..., 65, 66\}$. Notice that $g = 2$ is a generator of this group (that is, any element of $\mathbf{F}_{67}^*$ is a power of g).

Next Alice picks, say $x = 41$, and Bob chooses $y = 53$, keeping their respective values secret. Next, Alice computes $\mathbf{X} = g^x \bmod n = 2^{41} \bmod 67 = 12 \bmod 67$. Likewise, Bob

computes $\mathbf{Y} = g^y \bmod n = 2^{53} \bmod 67 = 41 \bmod 67$. Alice then sends Bob $\mathbf{X} = 12$ and Bob sends Alice $\mathbf{Y} = 41$.

Bob will then compute $k = \mathbf{X}^y \bmod n = 12^{53} \bmod 67 = 44 \bmod 67$, and Alice computes $k' = \mathbf{Y}^x \bmod n = 41^{41} \bmod 67 = 44 \bmod 67$. Thus, the secret shift-key is $k = k' = g^{xy} = 44$.

Alice and Bob have exchanged a key, without anyone else knowing the value of the key. However, the discrete logarithm problem would never be used to exchange a key with a system so insecure as shift-key encryption. Since the field $\mathbf{F}_{67}$ is so small, an outsider could easily find the discrete logarithm to the base 2 of 12 or 41 mod 67. Also, the shift-key encryption method does not have much security, and it would be pointless to use the discrete log on a system that could easily be broken otherwise [Koblitz 1994, 99-100]. The Diffie-Hellman key exchange algorithm would most likely be used on top of a private key algorithm, where the key is only to be known by two private parties.

4.3 Using the Discrete Logarithm as a method of message transmission

The discrete logarithm problem could also be used in message transmission. One such method of message transmission using the discrete log is the Massey-Omura cryptosystem.

The Massey-Omura Cryptosystem for Message Transmission

Assume that everyone has agreed upon a finite field, $\mathbf{F}_q$, which is fixed and publicly known. Each user of the system secretly selects a random integer e such that $0 < e < q\text{-}1$ and $\gcd(e,q\text{-}1) = 1$. Using the Euclidean Algorithm, each user computes the inverse $d = e^{-1} \pmod{(q\text{-}1)}$ (i.e. $de = 1 \bmod (q - 1)$).

If Alice wants to send a message P to Bob, then she must first send him the encrypted message P^{e_A}. This value does not hold any meaning for Bob, since he does not possess the decrypting key d_A, and therefore cannot recover the message P.

Bob will now take the message P^{e_A} received from Alice, and raise it to the e_B (Bob's secret encryption key) to get $P^{e_A e_B}$ and sends this back to Alice. Now Alice raises this message to her decryption value d_A, yielding $(P^{e_A e_B})^{d_A} = P^{e_B}$. This is in turn sent back to Bob, who can now decipher the message with his decryption key d_B, giving $(P^{e_B})^{d_B} = P$.

"The methodology behind this system is rather simple, and can be generalized to settings where one is using other processes besides exponentiation in finite groups." [Koblitz 1994, 100] However, this system can have some problems.

Suppose that when Alice sent the message P^{e_A}, it was intercepted by an outside user, C. This outsider, using his/her own private encryption key, e_C, could compute $P^{e_A e_C}$ and return it to Alice. When Alice received the message, she would not know that it did not come from Bob. If she decrypted the message with her decryption key, and sent P^{e_C}, user C could decrypt the message, which was not intended for him/her in the first place. For this reason a good signature scheme must be used with this cryptosystem to assure to Alice that the person who returned the message was Bob, and not some outside user. A final problem with this method is lack of efficiency. With so many transactions, it would take longer for Bob to receive the message using this system rather than others.

4.4 Comments on the Discrete Logarithm

Although it is very hard to find discrete logs in finite fields, it is not impossible. On page 102 of Neal Koblitz's book, *A Course in Number Theory and Cryptography*, a method is given by which one can find discrete logs in finite fields. The process is rather

long and complex, and I encourage the interested reader to consult this section of Koblitz's book (source number 11 in the bibliography).

5.0 Methods of determining primality

In recent years fast primality testing algorithms have been a popular subject of research for the last few decades. Some of the modern methods have been incorporated into computer algebra systems as standard, making it easier for one to tell if a number is prime (or probably prime) [Pinch 1993, 1203].

However, determining if a number is truly prime takes a long time as numbers get large. So called 'perfect' primality tests are those that determine if a number is *definitely* prime, where many tests can only determine the probability of a number being prime. Methods that determine a number is 'probably' prime are usually faster than the perfect primality tests, and nearly as effective. One such method which determines that a number is probably prime is Miller's test.

Described in the following sections is one old 'perfect' tests for determining primality: trial division. A second (though unreliable) test which can tell one if a number is prime is Fermat's test. Miller's test is also explained.

5.1 Trial division

The first and obvious algorithm that one can use is trial division. That is, given an integer n, try dividing n by all integers from 2 up to the square root of n to see whether any are factors of n. If one is found, then n is composite; if not, then n is prime. [Pinch 1993, 1203] The problems with this test is the amount of time that it takes to determine

whether or not n is prime, which is in the worst case of the order of the square root of n. If one were trying to factor a 100-digit number, this would not be the method to use.

5.2 The Fermat test

The Fermat test is a rather old primality test which relies upon the contrapositive of Fermat's Theorem[3]. Suppose we were trying to test a number n for primality. We must first check if there is an a such that $1 < a < n$ and $a^{n-1} \neq 1 \bmod n$. If so, then n must be composite. Since a^{n-1} can be computed modulo n in about log n modulo n multiplications, this condition is very fast to check. The value a is called the *base* for this test. [Pinch 1993, 1204]

However, if we find that $a^{n-1} \bmod n$ is 1, then we cannot conclude that n is prime from this test, and we must try again using a different base a. Another problem is that of Carmichael numbers, which are composite yet pass this test for all bases to which they are relatively prime. [Bressoud 1989, 33] Using the Fermat test, we could mistakenly think that one of these Carmichael numbers is prime.

5.3 Miller's test

The two previous algorithms can be used to determine whether a number is prime (with some degree of error). Miller's test takes a different angle and can be used to determine that a number is probably prime. The test is also called the strong pseudoprime test (rarer than pseudoprimes[4]), and every composite number that passes Miller's test is called a strong pseudoprime.

---

[3] Fermat's Theorem: If p is prime then, for $1 \leq a < p$, we have $a^{p-1} \equiv 1 \bmod n$.

[4] A *pseudoprime* is a composite number which fails a test to tell whether it is composite (e.g. Fermat's test). It can be explained as a composite that "acts like" a prime.

Miller's test works as follows. Suppose we want to see if an integer n is prime. We start with $n - 1 = 2^a t$, where $a \geq 1$ and t is an odd integer. Next, we let b be any integer with $\gcd(b, n) = 1$. We say that n passes Miller's test for base b if $b^t = \pm 1 \bmod n$ or $b^{2^j t} = -1 \bmod n$ for some j such that $1 \leq j \leq a - 1$.

For example, to determine whether the number 89 is prime, we first calculate $89 - 1 = 88 = 2^1 \cdot 44$. Now we have $a = 1$ and $t = 44$, and $2^{44} = 1 \bmod 89$. Thus, 89 passes Miller's test for $b = 2$, and is probably prime. In another example, using Miller's test on 2047 will show $2047 - 1 = 2046 = 2^1 \cdot 1023$. Therefore, $a = 1$ and $t = 1023$, and $2^{1023} = 1 \bmod 2047$, so it passes Miller's test. However, 2047 is composite ($23 \cdot 89$), and we say 2047 is a strong pseudoprime.

In the second example we see that Miller's test does have its faults. Strong pseudoprimes could trick one into thinking he/she has a number that is probably prime. We can reduce the risk of being fooled like this by re-testing the number for a different base b when it passes on the first base. This will increase the probability that the number is prime.

Miller's test comes into determining true primality of numbers through the generalization of the Riemann hypothesis. It is defined as: If the generalized Riemann hypothesis is true, then if n is a strong pseudoprime for base b for all integers a with $1 < a < 2(\log n)^2$, then n is prime. The Riemann hypothesis is very complex, and interested readers can consult a description in the article *On the Zeros of the Riemann Zeta Function in the Critical Strip. I.* Details about this article can be found in the bibliography [Brent, 1979].

6.0 Methods of factoring numbers

We learned from section three that in order to break the RSA algorithm, one must factor n. Once the factors are found, it is easy to calculate the decryption key. We will now look at a couple of different factoring algorithms that can be used to attempt such a task. The first method is the Quadratic sieve, in which we determine the factors of n through m-smooth numbers near the square root of n.

Perhaps the most effective factorization algorithms are a variation of a sieve, including the number field sieve, the multiple polynomial quadratic sieve, and different variations of each of these. Other methods include Continued Fractions, Dixon's algorithm, Euler's method, as well as many others. In this paper we will concentrate only on the Quadratic Sieve.

The first method uses what is called a *factor base*, which is a set of numbers from which a factor of a number n could possibly be found. The factor base consists of prime numbers x which are quadratic residues modulo n. A number x is a *quadratic residue* mod n if there exists an integer $t < n$ such that $t^2 = x \bmod n$. Also, the quadratic sieve bases its factoring of large numbers on smaller numbers which are m-smooth, i.e. numbers all of whose prime factors are less than m.

The Quadratic sieve

Created in 1980 by Carl Pomerance, the quadratic sieve is one well-known method of factoring large numbers. In using the quadratic sieve, one starts out with the number n to factor. Next, a relatively small number m is chosen. The factor base for n, or set of numbers to try to factor n with, is now computed and consists of all of the primes p less than m for which p is a quadratic residue modulo n.

Next, a table is formed with two columns. The first is a consecutive list of values x around the square root of n. The second column is the value of $x^2 - n$ for the values of x in the first column.

Once the table has been formed and the factor base determined, then all of the numbers in the $x^2 - n$ column are tested to see if they are m-smooth. When all of the m-smooth numbers have been found, then the prime factorization of each is taken. We then look for two or more numbers that make a perfect square (mod n) when they are multiplied together, which occurs when the prime factorization of these numbers contains prime numbers to an even power. If a combination is found, then the numbers (not the prime factors) are multiplied together mod n, and yield a result y. The prime factorization of all the numbers multiplied together we will call k. Then, we can write the equation:

$y^2 = (\sqrt{k})^2 \bmod n$, and if $\gcd(y - \sqrt{k}, n) \neq 1, n$, then the result of $\gcd(y - \sqrt{k}, n)$ is a factor of n.

A variation on this method has proven to be efficient in determining the factors of very large numbers, factoring a 129-digit number in 1994 [Pomerance 1996, 1473].

Let's look at an example using the quadratic sieve. Suppose we want to factor a number $n = 6731$. First we determine the square root of 6731, which is roughly 82, and also decide on an m-value of 17. Next, we choose values around 82 with which we will try to factor n, say from 71 to 97. For each of the values x in the factor base, we compute $x^2 - 6731$ and find the $\log_2(x^2 - 6731)$. When this is all complete, we can fill in the first three columns of table 1 (p. 22).

Now, for each prime p in the factor base (1 through 17) such that the Jacobi symbol[5] of 6731 over p is equal to 1, we find solutions to $t^2 = 6731 \bmod p$. The possible values for p include 2, 5, 7, 13 and 17. If it happens that $x = t \bmod p$, then p divides $(x^2 - 6731)$, and thus p divides every p^{th} term in the list after x. So, we subract $\log_2 p$ from the previously calculated log on the list.

Table 1 : Quadratic Sieve for n = 6731

x	x^2 - 6731	log 2	2	5	5^2	7	7^2	13	13^2	17
71	-1690	11	10	7				3	-1	
72	-1547	11				8		4		-1
73	-1402	11	10							
74	-1255	11		8						
75	-1106	11	10			7				
76	-955	10		7						
77	-802	10	9							
78	-647	10								
79	-490	9	8	5		2	-1			
80	-331	9								
81	-170	8	7	4						-1
82	-7	3				0				
83	158	8	7							
84	325	9		6	3			-1		
85	494	9	8					4		
86	665	10		7		4				
87	838	10	9							
88	1013	10								
89	1190	11	10	7		4				-1
90	1369	11								
91	1550	11	10	7	4					
92	1733	11								
93	1918	11	10			7				
94	2105	12		9						
95	2294	12	11							
96	2485	12		9		6				
97	2678	12	11					7		

[5] The Jacobi Symbol ((n/m)) for odd m > 1 is defined as the product of the Legendre symbols $((n/p_i))$, where the product is over all primes p_i dividing m (including multiplicities). The Legendre Symbol ((n/m)) for prime m checks whether or not n is a quadratic residue modulo m. The Legendre symbol returns 0 if m divides n, -1 if n is not a quadratic residue, and 1 if n is a quadratic residue.

As you can see in the table, the $\log_2$ calculations are made for each value x. For the next column we use the prime 2 from within the factor base, and find the solution to $t^2 = 6731 \bmod 2$ to be 1. So, we find the first x that is 1 mod 2 (which turns out to be 71), and subtract $\log_2 2 = 1$ from every second base starting with the first. Now we go onto the next prime number in the factor base (5) and find that the solutions to $t^2 = 6731 \bmod 5$ to be 1 and 4 mod 5. The first x which is 1 mod 5 is 71, so we subtract $\log_2 5$, which rounds up to 3, from each of the bases. The process repeats with 5^2, 7, 7^2, ... , up to 17.

After this has been done for all of the values, we see that some of the logs in the furthest columns to the right are less than or equal to zero. Every x associated with a value less than or equal to zero are m-smooth, and we now know the factorization of each:

$71^2 - 6731 = -1 \cdot 2 \cdot 5 \cdot 13^2$
$72^2 - 6731 = -1 \cdot 7 \cdot 13 \cdot 17$
$79^2 - 6731 = -1 \cdot 2 \cdot 5 \cdot 7^2$
$81^2 - 6731 = -1 \cdot 2 \cdot 5 \cdot 17$
$82^2 - 6731 = -1 \cdot 7$
$84^2 - 6731 = 5^2 \cdot 13$
$89^2 - 6731 = 2 \cdot 5 \cdot 7 \cdot 17$

Now that we know these values, we set a matrix with the columns representing the x values and the rows representing their factors. Each 1 entry signifies that the factor has odd multiplicity in the factorization. For example, by looking at the matrix below, we can see that 71 has an odd number of the factors -1, 2 and 5.

Fig. 2 : Vectors

	71	72	79	81	82	84	89			71	72	79	81	82	84	89
-1	1	1	1	1	1	0	0			1	0	1	0	0	0	1
2	1	0	1	1	0	0	1			0	1	0	0	0	0	1
5	1	0	1	1	0	0	1		RREF	0	0	0	1	0	0	0
7	0	1	0	0	1	0	1		⇒	0	0	0	0	1	0	0
13	0	1	0	0	0	1	0			0	0	0	0	0	1	1
17	0	1	0	1	0	0	1			0	0	0	0	0	0	0

In the matrix on the right we can see that the 71 and 79 columns are the same. By this we know that $(71^2 - 6731)(79^2 - 6731) = (-1)^2 \cdot 2^2 \cdot 5^2 \cdot 7^2 \cdot 13^2$ is a perfect square. Thus, $(71^2 \cdot 79^2) = 5609^2 = (2 \cdot 5 \cdot 7 \cdot 13)^2 \bmod 6731 = 910^2 \bmod 6731$. Finally, we see that gcd(5609 - 910, 6731) = 127, and therefore know that 127 and 53 are factors of 6731.

7.0 Programming the RSA algorithm

The final portion of this project was to implement the RSA algorithm in the C++ programming language. Due to the scope of the project and the time to complete the project, a full implementation would not be possible. Instead, a prototype of the RSA algorithm in which the size of the inputs would be restricted, would be implemented.

7.1 Layout of the program

There are two different classes (Explained in the following section), and one executable program, RSA.C. The steps in the program are as follows:

- Randomly generate prime numbers p and q.
- Compute $n = pq$ and $\varphi(n) = (p - 1)(q - 1)$.
- Randomly generate e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
- Compute $d = e^{-1} \bmod \varphi(n)$.
- Get the message and change it to numeric form.
- Encipher this message.
- Decipher the message.
- Translate the message back to its alpha-numeric form.

To accomplish all of these steps, the following functions are needed:

- A random number generator for the numbers p, q, and e.
- A function to make sure that p and q are prime. (There are many primality tests which can be used in this step)
- A function to compute n, $\varphi(n)$, and d. These are straight calculations.
- A function to translate a message into numeric form.
- A function to encipher a message.
- A function to decipher a message.
- A function to translate a message from numeric to alpha-numeric form.

All of these functions are explained in the following section.

7.2 Classes

There are two classes (.h files) used in this program: RandGen and NumberFuncs.

7.2.1 Class RandGen (Rando.h) [Budd 1994, 380]

This class is designated as the random number generator class. The function RandInt is that which generates the number. The range is specified when calling the Generate function. For the values p and q, the chosen range is 100 to 200, and for e the range is from 200 to 500.

```
class RandGen
{
  public:
    RandGen();                                  // set seed for all instances
    int RandInt(int max = INT_MAX);             // returns int in [0..max)
    int RandInt(int low, int max);        // returns int in [low..max]
  private:
    static int initialized;                     // for 'per-class' initialization
};
```

7.2.2 Class NumberFuncs (NumberFuncs.h)

Since there are many algorithms that must be applied to different values, I have put together a class which will host the computational functions necessary for the prototype of the RSA algorithm:

```
class NumberFuncs
{
      public:
        NumberFuncs();                  // Constructor
        int Generate(int min, int max); //return the random number generated.
        int GCD(int a, int b);          // check if two numbers are relatively prime
        int MakePrime(int x);           //Make the integer Prime
        int EncryptKey(int phi);        // Make the encryption key
        int DecryptKey(int e, int phi); // Calculate the Decryption key.
        int FindNumeric(char x);  // Find the numeric value of a character
        char FindAlpha(int y);      // Find the alphanumeric value of an integer.
        int Encrypt(int w, int x, int y, int e, int n); // Encrypt a numeric value
        int Decrypt(int msg, int numchar, int d, int n); // Decrypt a numeric value
        int Powermod(int base, int exponent, int mod);  // for the encryption
private:
       RandGen myGenerator;         //random number generator
};
```

The Generate function calls the RandInt function from class RandGen. The MakeOdd and MakePrime functions are used on the number generated to make the number generated prime, as is needed with the values p and q. The EncryptKey and DecryptKey functions generate the encryption and decryption keys.

FindNumeric is a function that translates an alpha message into its numeric value (which is necessary for the encryption process to occur). The FindAlpha function is used in the decryption process to find the alpha value of a numeric message. Encrypt and Decrypt do just what you might expect them to: encrypt and decrypt the message. Finally, Powermod is a function which takes a base to a power mod some value. For additional references, see NumberFuncs.C in Appendix A.

7.3 The main program (RSA.C)

This is the part of the program that uses all of the functions in section 7.2 to implement the RSA algorithm. The first block of code is the initializing of the objects and variables that will be needed throughout the program. There is an object of type NumberFuncs for the numerical calculations and translation/encryption/decryption functions on the message.

```
#include <math.h>^M
#include <iostream.h>
#include "NumberFuncs.h"

main()
{
    NumberFuncs prime;    // Object of type NumberFuncs
    int p;
    int q;
    int n;        // integer for storing numbers
    int e;        // encrypting key
    int d;        // decrypting key
    int numchar;        // to signify the number of characters in a block..
    char plaintext[100];  // plaintext message
    int ciphertext[100];  // ciphertext message
    int emsg[50];         // encrypted message array
    char choice;          // choice for switch
    int numptx[100];      // plaintext message after decrypted.
    char chptx[100];      // to contain alpha value of decrypted message.

    p = prime.Generate(100, 200);     //Generate one digit
    q = prime.Generate(100, 200);     //Generate another digit
    p = prime.MakePrime(p);  // Make p prime
    q = prime.MakePrime(q); // Make q prime

    int phi = (p-1)*(q-1);  // phi of n
    n = p*q;              // n for encrypting and decrypting.

    e = prime.EncryptKey(phi);
    d = prime.DecryptKey(e, phi);

    cout << "Encryption and decryption will be done with the ";
    cout << "following values: " << endl;
    cout << "p: " << '\t' << p << endl;
    cout << "q: " << '\t' << q << endl;
    cout << "Phi: " << '\t' << phi << endl;
    cout << "n: " << '\t' << n << endl;
    cout << "e: " << '\t' << e << endl;
    cout << "d: " << '\t' << d << endl;
    cout << endl;
```

RSA.C (continued)

```
// Obtain a word and put into plaintext array.
cout << "What word would you like to encrypt? " << endl;
cin >> plaintext;

// Find the numeric form of the message.
for(int i=0; plaintext[i]!= '\0'; i++)
{
   ciphertext[i] = prime.FindNumeric(plaintext[i]);
   cout << ciphertext[i] << " ";
}


// fill the rest of the array with negative 1's to signify there aren't
// any other values in the array...
 for(int k=i; k<100; k++)
 {
      ciphertext[k] = -1;
 }

 cout << endl;

/****************************************\
 * Encrypt the message.                 *
\****************************************/

 i = 0;  // counter for emsg[i]

 for(int j=0; ciphertext[j] != -1; j = j+3)
 {
   // encrypt three characters at a time.
   emsg[i] = prime.Encrypt(ciphertext[j], ciphertext[j+1],
      ciphertext[j+2], e, n);

   if(ciphertext[j+1] == -1)
   {
     numchar = 1; // signifies one character in the block
   }
   else if(ciphertext[j+2] == -1)
   {
     numchar = 2; // signifies two characters in the block
   }
   else numchar = 3; // otherwise three characters in the block.

   i++;
   emsg[i] = numchar;       // to signify the number of characters in
                            // this block.....
   i++;
 }

 emsg[i] = -1; // null value
```

RSA.C (continued)

```
// print out the ciphertext message that is sent.
for(j=0; emsg[j]!=-1; j=j+2)
{
    cout << emsg[j] << "00";
}
cout << endl;

/************************************************\
 *  DECRYPTION                                    *
\************************************************/

cout << "************************************************" << endl;
cout << "************************************************" << endl;
cout << endl << "Would you like to decrypt the message? ";
cin >> choice;

if((choice == 'y') || (choice == 'Y'))
{
  j = 0; // count for ptx array.

  for(i=0; emsg[i]!=-1; i= i+2)
  {
     // dummy is a pointer to the temp array in Decrypt.
     int * dummy = prime.Decrypt(emsg[i], emsg[i+1], d, n);

     // Insert the values into the numptx array.
     for(int k=0; k<3; k++)
     {
              numptx[j] = dummy[k];
              j++;
     }
  }

  numptx[j] = -1;  // insert end of array value.

  // Translate to alpha...
  for(i=0; numptx[i]!=-1; i++)
  {
     chptx[i] = prime.FindAlpha(numptx[i]);
  }

  chptx[i] = 26;   // insert a null value at the end of the array.

  cout << endl << "The factorization of each block yields: " << endl;

  // print out the numeric form
  for(i = 0; numptx[i]!=-1; i++)
  {
     cout << numptx[i] << " ";
  }

  cout << endl << endl << "Translated to alpha form yields: " << endl;
```

RSA.C (continued)

```
    // print out the character plaintext.
    for(i=0; chptx[i]!=26; i++)
    {
        cout << chptx[i];
    }
  }
  else if(choice == 'n' || choice == 'N')
  {
    // the program is ended.
    cout << "Have a nice day!! " << endl;
  }

  cout << endl;
}
```

7.4 Running the program

When the program is executed, the random number generator is first called to generate the values for p and q. If these values are not prime, then they are made prime in the MakePrime function by trial division. After it is determined that the p and q values we have are prime, then the values of n and $\varphi(n)$ are calculated and a value for e is generated which is relatively prime to $\varphi(n)$. The generation of the value e is done in the EncryptKey function. The DecryptKey function is used to find a d such that $de \equiv 1 \mod \varphi(n)$.

Now that we have all of the necessary values for encryption and decryption of a message, a message is now requested. The word that is received is placed in the plaintext array, and three blocks of this array are sent to the Encrypt function. The Encrypt function first translates the values to their respective integer values, places the integer values in the equation $a \cdot 26^2 + b \cdot 26 + c$, where a, b and c are the values from first to last in the word. This results in yet another value, which is encrypted using the encryption key (e, n). The encrypted message is returned into the ciphertext array.

After the entire word is encrypted, the ciphertext is printed out, which consists of each of the values returned by the Encrypt function in order, padded with zeroes to distinguish the end of a block.

The user is asked whether or not they would like to decipher the message. If a "no" answer is indicated, the program is finished. If the user indicates "yes", then the Decrypt function is called. This will decrypt the function using the values d, n, and the message. The result is then factored and the results are placed in an array. A pointer to this array is returned, and the values are copied into the numptx array. After the entire message has been decrypted, it is translated back into alpha form. Both the numeric and alpha form are printed out.

7.5 Problems with RSA.

There were some problems that had to be faced when programming an algorithm such as this one. First of all, since the C++ language doesn't handle integers over 10 digits long very efficiently, it would be rather difficult to fully implement the RSA model. The number generator alone would have to be able to generate digits over 130 digits long, which is quite a step up from ten.

The next problem that had to be dealt with was the method of translating the plaintext message to a numerical form. There are quite a few ways that this can be done, some of which are quite complex. I decided to take the easy, though inefficient route of using a switch to find the numerical values for individual characters. The drawback with this is that if you want every character (including such things as semi-colons and brackets), you will have to have one line for each character to be translated.

Trying to determine the best way to encipher the message was another problem. How many blocks should I encrypt at a time? This was a big factor, since the more characters that were taken at one time, the larger the number generated in the translation and encryption process. And with the 10-digit efficiency imposed, this number had to be less than 4.

8.0 Conclusion

The study of cryptography is unique because of its interrelation with both math and computer science. The theory behind cryptography involves the use of number theory, while the application of it involves different aspects of computer science. For instance, with the theory of cryptography, the primality or pseudoprime tests such as Fermat's or Miller's are used in some cryptosystems. On the other hand, the application of cryptography involves components of computer science such as programming languages, network protocols and computational capabilities.

Although a major in Mathematics/Computer Science is offered at Saint John's University and the College of Saint Benedict, there are not any courses specified as being Math/CS. Therefore, it is hard to determine how these two concentrations come together in the real world. We can learn all about Euler's and Fermat's theorems in one class and about operating systems and networking in another, but never relate the two.

What most people don't realize is that we can use the knowledge of mathematical theorems and algorithms to draw conclusions or protocols in the world of computer science. If we study theory of computer science, we can see that most of the logic behind it is based on mathematical principles, not just programming or building machines.

Even with theory of math or computer science, how are we to know specifically where the two meet in the real world? That is, what applications in computer science depend on the proven mathematical theories? Cryptography is one example where mathematical algorithms are applied in computer science, with the RSA model being dependent on mathematical algorithms to find two prime numbers. Also, when one is using a word processor such as Microsoft Word, each character that is typed involves the use of mathematical algorithms to determine which pixels are 'activated' and which ones are not. Mathematics is so deeply embedded in computer science that it seems that the latter is somewhat of a child or subdivision of the former.

We can also look at this relation from the view of how computer science has helped make mathematicians' calculations easier by implementing the different algorithms. For example, in using *Mathematica* to find all of the divisors of an integer a, we only have to type "Divisors[a]" instead of using different mathematical algorithms to find the values. Even though computer applications have helped mathematicians, I remind you that the field of computer science is extremely young compared to that of math, and the application of mathematical theories and algorithms in the area of computer science is much more common.

There is another problem with relating the two subjects in this fashion. When we think of computer science, we think of different software packages such as Window's 95, Microsoft Excel, or Solitaire. In software engineering, most software today is developed with the help of other software, without any knowledge of math being required, making it difficult to determine how to make the connection between the two subjects. If we want to truly relate math and computer science, we must look deeper than the application level.

What I mean by this is that we must look at the architectural level of applications. For instance, what makes the applications work? Behind everyone's computer screen there are ASCII code and binary numbers. What we do not see are the mathematical algorithms which are applied to the binary numbers in order to make the application perform in the fashion that it does. Everything we do with computers can somehow be translated into a numeric form with the help of math.

With the topic of cryptography, one of my goals was to determine and address the issues which exist in the relationship between math and computer science. Some of these include security and efficiency of cryptosystems, computational feasibility of implementing cryptosystems, and government encryption standards.

For instance, some issues with the RSA algorithm included knowing that a decryption key existed, and finding out why it can exist. The existence of a decryption key in the RSA model can't be proven without the help of Fermat's theorem and Euler's generalization.

A second issue is related to the security of different cryptosystems. Is it easy to break a specific cryptosystem? What are some of the methods that can and have been used to break them? Again, with the RSA model, in order to break the system we must find the prime factors of n. Different methods such as the Quadratic sieve and Dixon's algorithm can be used in order to find these factors. The issue lies in the time that it would take one to break the a cryptosystem using one of these methods. The faster a cryptosystem can be broken, the higher the risk to the user.

Other issues reside in the area of computer science, most of which are the result of the mathematical computations involved in the algorithms. To many it would seem that any

algorithm could be implemented with the use of computer science. This is almost true. First of all, in implementing a mathematical algorithm with the use of computers, the values given are not always accurate. It is possible to calculate a value to a certain preciseness, but it is nearly impossible to capture the true value. For example, since the expansion is infinite, it is completely impossible to obtain the entire expansion of pi.

Another example where the limits of computer capabilities are an issue is in the implementation of the RSA model. Implementing this algorithm in a truly secure fashion involves the generation of the primes p and q at a size of 130 digits or more. However, computer languages such as C++ do not handle numbers over ten digits very effectively. There are ways to get around this problem through additional programming, but deeper knowledge of UNIX and C++ is necessary.

Computing time is another issue when implementing the algorithms. Again with the RSA model, prime numbers at least 130 digits in length are necessary in order to provide adequate security. However, as the size of the number increases, the more difficult it is to find prime numbers. Therefore, even with the use of a random number generator, it could take a large amount of computing time to find a prime number.

Even after we have generated the prime numbers, it is possible that someone else could use similar algorithms to generate prime values which are the same as ours, thus having the exact same encryption and decryption keys. This is an issue that needs special concern, since it is not against the law to simply make some mathematical calculations.

An issue that I was not able to address was that of government regulations and standards imposed on encryption algorithms. Government officials might argue that cryptography can be used by criminals or terrorists in planning their operations. The

government wants to know everything that is going on in the world, and cryptography limits this knowledge. Government would like to limit the level of cryptography that is used, where it is used, and who it is used by. However, the security of innocent users would be at stake if the government were to impose regulations against public use of cryptography.

Overall, the area of math is extremely large, and has been studied since the time of the Egyptians. The study of computer science is young, yet growing at an extremely high rate due to an ever-increasing level of technology. However, even though math and computer science may look entirely different from an outside view, they are deeply related to one another. Moreover, with cryptography, this relationship must remain strong in order to be able to provide adequate security of information over networks.

9.0 Appendix A : Programs

9.1 NumberFuncs.C

```
#include <math.h>^M
#include <iostream.h>
#include "NumberFuncs.h"

NumberFuncs::NumberFuncs()
{
}

int NumberFuncs::Generate(int min, int max)
{
    int x = myGenerator.RandInt(min, max);  //in range [1...range]

    if(x%2 == 0)
    {
      x=x+1;
      }
    return x;
}

int NumberFuncs::GCD(int a, int b)
{
    int temp;

    while(b != 0)
    {
      temp = b;
      b = a%b;
      a = temp;
    }
    return a;
}

int NumberFuncs::MakePrime(int x)
{
    int primetester = 3;
    int isprime = 0;  // 0 means not prime, 1 means prime.
    int square = sqrt(x);

    while(isprime == 0)
    {
      if(primetester <= square)
        {
```

NumberFuncs.C (continued)

```
        if(x%primetester == 0)
          {
              primetester = 3;
              x = x+2;
              square = sqrt(x);
          }
        else
        {
              primetester = primetester+2;
         }
      }
      else
      {
        isprime = 1;
      }
    }
    return x;
}

int NumberFuncs::EncryptKey(int phi)
{
    NumberFuncs myNumber;     // object of type NumberFuncs

    int e = myNumber.Generate(200, 500);

    if(e%2 == 0)
    {
        e = e + 1;      // if e is even, make it odd.. this is to increase the chance
                        // of finding a value that is relatively prime to phi of n.
    }

    while(myNumber.GCD(phi, e) != 1)
    {
      e = e + 2;        // increase e until you find one that is relatively
                //  prime to phi of n.
    }
    return e;
}

int NumberFuncs::DecryptKey(int e, int phi)
{
    // we are looking for a value d such that (de = 1 mod phi(n)).
    // the easiest way to find this is to start at the beginning and
    // work our way up. That is, set d = 1 and see if the case is true.
    // If not, increment d by 2 (because d*e must be odd since phi(n) is
    // even, and also since e is always odd (because gcd(e,phi) = 1)).
    // With an odd d and e, we will have an odd result when multiplying
    // them together.
```

NumberFuncs.C (continued)

```
    int d = 1;
    int mod = (d*e)%phi;
    while(mod != 1)
    {
      d = d+2;
      mod = (d*e)%phi;
    }
    return d;
}

int NumberFuncs::FindNumeric(char x)
{
    return (x-'a'+0);
}

char NumberFuncs::FindAlpha(int y)
{
    return (y+'a');
}

int NumberFuncs::Encrypt(int w, int x, int y, int e, int n)
{
    NumberFuncs myNumber;
    cout << "first letter =  " << w << endl;
    cout << "second letter = " << x;

    if(x == -1)
    {
        cout << " (no character)";
    }
    cout << endl << "third letter =  " << y;

    if(y == -1)
    {
        cout << " (no character)";
    }
    cout << endl;

    int a = 26*26;
    int b = 26;
    int c;

    if(x == -1) // if the second value is a zero
    {
      c = a*w;
      y = 0;
    }
```

NumberFuncs.C (continued)

```
    else if(y == -1)
    {
      c = a*w + b*x;
    }
    else
    {
      c = a*w + b*x + y;
    }

    cout << "The numeric form of this block is " << c << endl;

    // take emsg to the eth power mod n to encrypt
    int emsg = myNumber.Powermod(c,e,n);
    cout << "The encrypted message yields " << emsg << endl;
    return emsg; // supposed to return emsg

}

int * NumberFuncs::Decrypt(int cptxt, int numchar, int d, int n)
{
    // This function takes in a block of the ciphertext message, the
    // number of characters in the block, and the decryption key (d,n).
    // The block is first deciphered, then factored to find the numeric
    // value for each letter.  The values are collected in an integer
    // array and then a pointer to that array is returned to the main
    // program.
    NumberFuncs myNumber;     // object of type NumberFuncs.
    int a, b, c;          // These are for the numeric character values.

    int * temp = new int[3];  // An array for the character values.

    // Decipher the message block.
    int msg = myNumber.Powermod(cptxt,d,n);

    // print out the decryption process
    cout << endl << "The ciphertext " << cptxt << " is decrypted: " << endl;
    cout << cptxt << "^" << d << " mod " << n << " = " << msg << endl;

    if(numchar == 1) // there is only one letter in the block..
    {
        a = (msg-(msg%(26*26)))/(26*26);

        temp[0] = a;  //The first letter in the block..(and only)
        temp[1] = -1; // No second or third letter.  insert -1 to
        temp[2] = -1; // signify that there is none.  We can't use
                      // '\0' because that is reserved for the
        return temp; // character 'a'.
    }
```

NumberFuncs.C (continued)

```
    else if(numchar == 2)  // There are only two values to be calculated
    {
        a = (msg-(msg%(26*26)))/(26*26);
        msg = msg%(26*26);
        b = (msg-(msg%26))/26;

        temp[0] = a;  // The first and second characters exist.
        temp[1] = b;  // there is no third character, so insert a
        temp[2] = -1; // -1 to signify so.

        return temp;
    }
    else        // there are three values to be calculated..
    {
        a = (msg-(msg%(26*26)))/(26*26); // factor by 26^2
        msg = msg%(26*26);               // The remainder.
        b = (msg-(msg%26))/26;           // divide by 26
        //msg = msg%(26*26);             // The remainder
        msg = msg%26;
        c = msg;

        temp[0] = a;
        temp[1] = b;
        temp[2] = c;

        return temp;
    }
}

int NumberFuncs::Powermod(int base, int exponent, int mod)
{
    int temp = base;

    if(exponent == 0)
    {
        return 1;
    }
    else
    {
        while(exponent != 1)  // != 1 because when 1 we are done.
        {
                base = (base * temp)%mod;
                exponent = exponent - 1;
        }
    }
    return base;
}
```

9.2 Rando.C [Budd 1994, 380]

```
#include <time.h>              // for time()
#include <limits.h>            // for INT_MAX
#include "Rando.h"

extern "C"
{
   long random();
   void srandom(int);
}

int RandGen::initialized = 0;

RandGen::RandGen()
// postcondition: system srandom used to initialize seed
//                once per program
{
   if (0 == initialized)
   {
      initialized = 1;        // only call srandom once
      srandom(int(time(0)));  // randomize
   }
}
int RandGen::RandInt(int max)
// precondition: max > 0
// postcondition: returns int in [0..max)
{
   return random() % max;
}

int RandGen::RandInt(int low, int max)
// precondition: low <= max
// postcondition: returns int in [low..max]
{
   return low + RandInt(max-low+1);
}

double RandGen::RandReal()
// postcondition: returns double in [0..1)
{
   return RandInt() / double(INT_MAX);
}
```

10.0 Program outputs

jjflynn@norcia 5% RSA
Encryption and decryption will be done with the following values:
p: 127
q: 107
Phi: 13356
n: 13589
e: 205
d: 7753

What word would you like to encrypt?
cryptography
2 17 24 15 19 14 6 17 0 15 7 24
first letter = 2
second letter = 17
third letter = 24
The numeric form of this block is 1818
The encrypted message yields 1604
first letter = 15
second letter = 19
third letter = 14
The numeric form of this block is 10648
The encrypted message yields 3536
first letter = 6
second letter = 17
third letter = 0
The numeric form of this block is 4498
The encrypted message yields 5062
first letter = 15
second letter = 7
third letter = 24
The numeric form of this block is 10346
The encrypted message yields 7090
160400353600506200709000
****
****

Would you like to decrypt the message? y

The ciphertext 1604 is decrypted:
1604^7753 mod 13589 = 1818

The ciphertext 3536 is decrypted:
3536^7753 mod 13589 = 10648

The ciphertext 5062 is decrypted:
5062^7753 mod 13589 = 4498

The ciphertext 7090 is decrypted:
7090^7753 mod 13589 = 10346

The factorization of each block yields:
2 17 24 15 19 14 6 17 0 15 7 24

Translated to alpha form yields:
cryptography

Bibliography

1. Adelman, L. "On Breaking the Iterated Merkle-Hellman Public-Key Cryptosystem." *Proceedings of the 15th ACM Symposium on the Theory of Computing* 1983. 402-412.

2. Adelman, L. Rivest, R.L. Shamir, A. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*. February 1978.

3. Beth, Th. Frisch, M. and Simmons, G.J. *Public Key Cryptography: State of the Art and Future Directions*. Springer-Verlag: New York. 1992.

4. Brent, R. P.; Van Der Lune, J.; Te Riele, H. J. J.; and Winter, D. T. "On the Zeros of the Riemann Zeta Function in the Critical Strip I." *Mathematical Computations* 1979. 1361-1372

5. Bressoud, David M. *Factorization and Primality Testing*. Springer-Verlag: New York. 1989.

6. Budd, Timothy. *Classic Data Structures in C++*. Addison-Wesley Publishing Co. 1994.

7. Delaurentis, J.M.. "A Further Weakness in the Common Modulus Protocol for the RSA Cryptoalgorithm." *Cryptologia 8* 1984 253-259.

8. Foss, Ann Marie. *An Analysis of the RSA Cryptosystem*. North Dakota State University Senior Thesis: Not Published. 1986.

9. Guy, R. *Unsolved Problems in Number Theory*. Springer-Verlag, New York, NY. 1994.

10. Huxley, M. N.. *The Distribution of Prime Numbers*. Oxford University Press. 1972.

11. Koblitz, Neal. *A Course in Number Theory and Cryptography*. Springer-Verlag: New York. 1994.

12. Konheim, Alan G. *Cryptography, A Primer*. John Wiley & Sons. 1981.

13. Kranakis, Evangelos. *Primality and Cryptography*. John Wiley & Sons Ltd. 1986.

14. Kurtzman, Joel. *The Death of Money*. New York, NY: Simon & Schuster, 1993.

15. Loxton, J.H. *Number Theory and Cryptography*. Cambridge University Press. 1990.

16. Miller, G. L. "Riemann's Hypothesis and Tests for Primality." *Journal of Computer and System Sciences* 1976 300-317.

17. Patterson, Wayne. *Mathematical Cryptology for Computer Scientists and Mathematicians*. Rowman & Littlefield. 1987.

18. Pinch, R.G. "Some Primality Testing Algorithms." *Notices of the American Mathematical Society* November 1993.

19. Pomerance, Carl. "Cryptology and Computational Number Theory." *Notices of the American Mathematical Society*. 1989.

20. Pomerance, Carl. "A Tale of Two Sieves." *Notices of the Mathematical Society* December 1996 1473-1485.

21. Rabin, M. O. "Digitalized Signatures and Public-Key Functions as Intraceable as Factorization." *MIT Laboratory for Computer Science Technical Report 212*. 1979.

22. Salomaa, A. *Decision Problems Arising From Knapsack Transformations*. Acta Cybernetica, 1991.

23. Salomaa, A. *Public-Key Cryptography*. Springer-Verlag, New York, NY. 1990.

24. Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons Ltd.: New York. 1994.

25. Silverman, Robert D. "A Perspective on Computational Number Theory." *Notices of the American Mathematical Society* July/August 1991. 562-568.

26. U.S. Congress, Office of Technology Assessment. *Electronic Enterprises: Looking to the Future, OTA-TCT-600*. (Washington, DC: U.S. Government Printing Office, May 1994).

27. U.S Congress, Office of Technology Assessment, *Information Security and Privacy in Networkd Environments, OTA-TCT-606* (Washington, DC: U.S Government Printing Office, September 1994).

28. F. Wrixon. *Codes and Cyphers*. Prentice Hall, New York. 1992.